ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
Mención Computación

Implementación eficiente de redes neuronales autoorganizadas en CUDA C/C++.

Effitient implementation of self organizing neural netword in CUDA C/C++.

Realizado por
Shelley Gill
Tutorizado por
Enrique Domínguez Merino
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Diciembre 2015

Fecha defensa:
El Secretario del Tribunal

Resumen: En este documento se expondrá una implementación del problema del viajante de comercio usando una implementación personalizada de un mapa auto-organizado basándose en soluciones anteriores y adaptándolas a la arquitectura CUDA, haciendo a la vez una comparativa de la implementación eficiente en CUDA C/C++ con la implementación de las funciones de GPU incluidas en el Parallel Computing Toolbox de Matlab.

La solución que se da reduce en casi un cuarto las iteraciones necesarias para llegar a una solución buena del problema mencionado, además de la mejora inminente del uso de las arquitecturas paralelas. En esta solución se estudia la mejora en tiempo que se consigue con el uso específico de la memoria compartida, siendo esta una de las herramientas más potentes para mejorar el rendimiento.

En lo referente a los tiempos de ejecución, se llega a concluir que la mejor solución es el lanzamiento de un kernel de CUDA desde Matlab a través de la funcionalidad incluida en el Parallel Computing Toolbox.

Palabras claves:CUDA, Parallel Computing Toolbox, Matlab, problema del viajante de comercio, mapas auto-organizadas, memoria compartida.

Abstract: In this document I will explain an implementation of the travelling salesman problem using a personalized implementation of a self-organizing map basing it on past solutions and adapting them to the CUDA arquitecture, making at the same time a comparison of the effitient implementation in CUDA C/C++ with the implementation of the GPU functions included in the Parallel Computing Toolbox from Matlab.

The solution given reduces in almost a quarter the iterations necessary for getting a good solution of the problem mentioned, as well as the imminent mark up from the use of parallel arquitectures. In this solution I study the tie mark up that is achieved by using the shared memory in CUDA specifically, this being one of the best tools for gining effitiency.

When it comes to the execution times, we can conclude that the best solution es the execution of a CUDA kernel from Matlab through the functionalities included in the Parallel Computing Toolbox.

Keywords: CUDA, Parallel Computing Toolbox, Matlab, travelling salesman problem, self-organizing maps, shared memory.

Índice general

Capítulo 1

Introducción

En este documento se procederá a exponer tanto datos teóricos como prácticos para la realización de un mapa auto-organizado adaptado al problema del viajante de comercio y a la arquitectura de tarjetas gráficas.

Además se hará una comparativa tanto en eficiencia como en facilidad de implementación y calidad de los resultados de la implementación a más bajo nivel como es el caso de CUDA o de la implementación realizada en Matlab.

1.1. Antecedentes

En el contexto actual, nos encontramos con una colección de problemas NP que mediante el uso de algoritmos de aprendizaje se puede llegar a obtener buenas aproximaciones en un tiempo mucho más razonable. Esto ocurre con el problema del viajante de comercio, que usando una aproximación mediante una red de Kohonen adaptada al problema, puede conseguir una mejora notable en el tiempo de ejecución.

Además con la mejora de los componentes dedicados al procesamiento masivo de datos en paralelo, podemos mejorar el rendimiento más aún, obteniendo en muchos casos la mejora gratuita con una mejora de hardware.

1.2. Objetivos

Se pretende dar una solución aproximada al problema del viajante de comercio, utilizando las tecnologías gráficas aplicadas a una solución adaptada a una red de Kohonen.

También se hará una comparación entre las implementaciones realizadas para concluir sobre la implementación en Matlab de las librerías de CUDA.

Para ello se va a repasar los detalles teórico-prácticos que se han utilizado, y se darán detalles sobre la implementación y las adaptaciones de los algoritmos que se han seguido.

1.3. Trabajo futuro

Dado que el problema del viajante pertenece al conjunto de los problemas NP, y en este conjunto podemos reducir un problema a otro del mismo conjunto con tiempo lineal, este trabajo es muy interesante para llegar a una aproximación buena para muchos problemas.

1.4. Estructura de la memoria

El documento estará organizado en capítulos diferenciados por tema, a continuación se da una breve explicación de cada uno.

Se estudiarán las RNA generales, en más profundidad las RNA con aprendizaje no supervisado y haciendo especial hincapié en las redes de Kohonen o mapas auto-organizados.

Además se estudiará las razones por las que pueden o no ser paralelizadas las RNA y las ventajas e inconvenientes a las que puede llevar.

Las tecnologías de las que haremos uso serán CUDA C/C++ y Matlab junto con el Parallel Computing Toolbox, y más específicamente su implementación de funciones de GP-GPU. Se verán aquí detalles específicos de los lenguajes, los modelos de computación, las infraestructuras necesarias y datos de rendimiento. También se dará a conocer la arquitectura GP-GPU con especial interés en la tarjeta gráfica usada para el trabajo.

Se verá en detalle el problema de viajante y con mayor detenimiento la adaptación del mismo al mapa auto-organizado para su resolución, además de hacer comparativas tanto de rendimiento como de la calidad.

El mismo problema será implementado en las tecnologías descritas y los detalles específicos de la implementación serán descritas, junto con las distintas implementaciones posibles a raíz de ejemplos y decisiones tomadas.

Se intentará además comprobar la eficiencia de la implementación de Matlab del paralelismo en GP-GPU en comparación con la realizada a mano en CUDA C/C++. Además se estudiará la dificultad que tiene para el programador y la calidad de los resultados.

Finalmente se expondrán una serie de conclusiones finales, ramificaciones posibles, escalabilidad e integración en otros entornos.

Capítulo 2

Mapas auto-organizados

En el ámbito de las RNA podemos definir una multitud de topologías, diseños y definiciones matemáticas. Entre ellas podemos destacar, por interés en este documento, las redes neuronales no supervisadas, y entre estas los mapas auto-organizadas, o redes SOM por sus siglas en Inglés.

Estas redes mapean el conocimiento de una forma parecida al que ocurre en nuestros cerebros, con el aprendizaje por experiencia, tienen la capacidad de clasificar conocimientos según su parecido a los ejemplos ya conocidos hasta el momento, y además permite un aprendizaje gradual y constante.

Para la redacción de este capítulo me baso en los apuntes de clase de Modelos Computacionales redactados por José Muñoz Pérez [1].

2.1. Redes Neuronales Artificiales

Las redes neuronales artificiales, en adelante RNA, son un modelo computacional que pretende imitar el funcionamiento de las redes neuronales naturales que se encuentran en la naturaleza. Podemos encontrar una gran

variedad de estas, desde el perceptrón simple, hasta los algoritmos de deep-learning, pasando por redes supervisadas, no supervisadas, auto-organizadas, retro-alimentadas y un largo etc.

Este modelo computacional se dota de ser paralelizable, que consiste en unas unidades de proceso o neuronas, masivamente interconectadas. Estas conexiones se conocen como conexiones sinápticas y se organizan por capas conectadas entre sí. Estas neuronas tienen una función de activación que tienen como entrada las activaciones de la capa anterior y el peso sináptico ligado a estas.

2.1.1. Aprendizaje no supervisado en RNAs

El aprendizaje no supervisado, como su nombre indica, consiste en aprender de forma autónoma, mediante los datos facilitados, sin recurrir a su posterior corrección de las conclusiones. Esto nos permite realizar un aprendizaje del que no se sabe su solución, o sencillamente en un conjunto de datos demasiado grande, para ir refutando los datos uno a uno. De esta manera, tras el aprendizaje, tenemos un conjunto de datos auto-organizados, lo que hace de este tipo de RNA una muy buena para la detección de señales, reconocimiento de patrones o soluciones óptimas a ciertos problemas.

Podemos destacar tres modelos de sistema auto-organizado: las Redes Neuronales Competitivas no supervisadas y las Redes auto-organizadas o red de Kohonen, esta última siendo el principal enfoque de este documento.

Redes Neuronales Competitivas no supervisadas

Según podemos ver en los apuntes mencionados anteriormente [1], Una red competitiva está constituida por N sensores de entrada, M unidades

de proceso (neuronas artificiales), y conexiones entre cada sensor y cada unidad de proceso, de manera que la conexión entre el sensor j y la unidad de proceso i tiene asociado un valor wij. Para cada entrada recogida por los sensores solamente una unidad de proceso se activa, aquella que tiene el mayor potencial sináptico, que se le considera como la unidad ganadora.

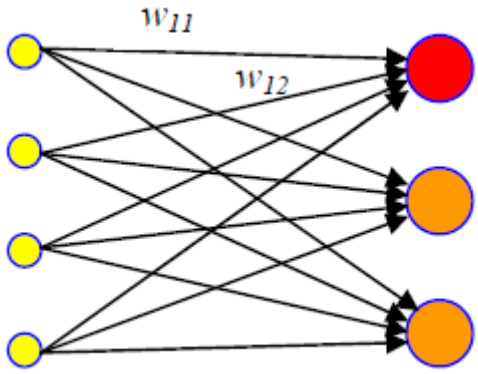

Figura 2.1: Representación de la arquitectura.

2.1.2. SOM o redes de Kohonen

En la topología natural del cerebro, se tiene en cuenta la posición de la neurona, a la hora de aprender, formándose así núcleos interconectados de computación. Las SOM tienen en cuenta, en alguna medida, esta topología, ya que cada neurona a la hora de hora de *aprender* o actualizar los pesos sinápticos que tienen asociado, también actualiza los pesos de sus vecinos, creando así unos núcleos en el espacio. Creado por Teuvo Kohonen en la Universidad de Tecnología de Helsinki en los años 80.

Para este propósito creamos una red con dos capas, la entrada o sensores y una rejilla normalmente rectangular de $M_1 x M_2$ unidades de proceso.

Además en este modelo se tiene en cuenta los vecinos a la hora de actualizar los pesos sinápticos, construyendo una plantilla mediante una función de vecindad como puede ser el correspondiente a la tabla 2.1.

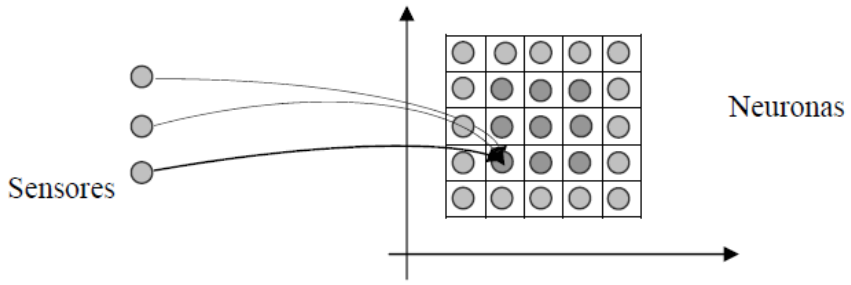

Figura 2.2: Representación del posicionamiento de las neuronas y sensores.

$$V(p_i, pr) = \begin{cases} 1 & si\ i = r \\ 0{,}5 & si\ |i - r| = 1 \\ 0 & en\ otro\ caso \end{cases}$$

Tabla 2.1: Función de vecindad.

Esta función de ejemplo da lugar a una rejilla que coincide con la tabla 2.2.

0	0.5	0
0.5	1	0.5
0	0.5	0

Tabla 2.2: Representación de la función de vecindad.

También es necesario escoger una tasa de aprendizaje tal que sea una función decreciente, ya sea lineal o no. Como ejemplo de una tasa linealmente decreciente podemos coger el siguiente donde $\eta_0 \in (0, 1]$ y T el número total de iteraciones, como en la tabla 2.3.

Tenemos también que actualizar los pesos sinápticos con la siguiente regla de aprendizaje, teniendo que w_i es el vector de pesos sinápticos designado a la neurona i, como en la tabla 2.4.

$$\eta(k) = \left(1 - \frac{k}{T}\right)\eta_0$$

Tabla 2.3: Función linealmente decreciente para la tasa de aprendizaje.

$$w_i(k+1) = w_i(k) + \eta(k)V(p_r, p_i)(x(k) - w_i(k)), i = 1,2,...,M_1 \times M_2$$

Tabla 2.4: Regla de aprendizaje.

2.1.3. Adaptaciones al problema del viajante

Dada la importancia académica de este problema he decidido implementar una adaptación de este problema basándome en la incluida en los apuntes [1] y en una implementación realizada en la Universidad de Nis, para optimizar el camino para recorrer los contenedores de residuos plásticos en la ciudad de Nis, usando sus coordenadas [2].

Adaptación incluida en los apuntes

En esta adaptación se tiene como entrada una serie de puntos que constituyen el mapa que se quiere optimizar, cambiamos la topología de disposición de las neuronas por una circular tres veces el número de entradas. Obteniendo una capa de neuronas dispuestas como en la figura 2.3.

Como se puede observar en la figura 2.3 la función de vecindad también sufre un cambio ya que este considera vecinos a las dos neuronas a cada lado.

Adaptación para la ciudad de Nis

Esta implementación cambia el planteamiento de la red del apartado anterior, eliminando la necesidad de tener en cuenta los pesos sinápticos,

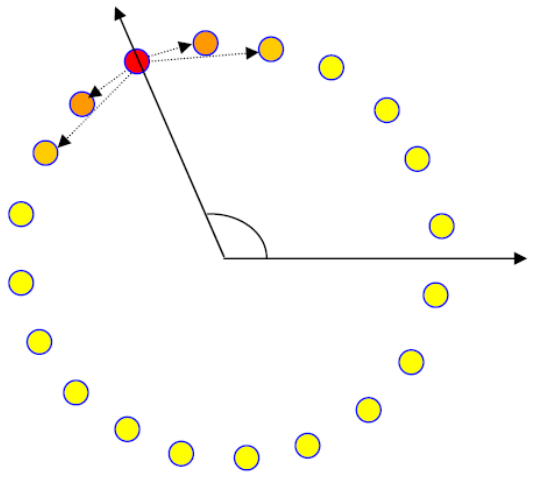

Figura 2.3: Representación de la arquitectura.

y realizando el aprendizaje mediante el movimiento de la posición de las neuronas en el espacio.

En este caso la entrada consiste en dos sensores que toman una lista de coordenadas, normalizadas según la fórmula $x_{scaled} = \frac{(x-x_{min})}{(x_{max}-x_{min})}$, de una a una, escogiendo como neurona ganadora la que se encuentra más cercana de la entrada. El mapa que se emplea es parecida al apartado anterior, las neuronas se encuentras dispuestas en un círculo de radio 1 y centro 0. Se tienen como vecinos las neuronas inmediatamente a cada lado.

La regla de aprendizaje queda como sigue:

$$f(\sigma,d) = e^{\left(\frac{-d^2}{\sigma^2}\right)}$$
$$y_j^{new} = y_j^{old} + \alpha \cdot f(\sigma,d) \cdot \left(x_i - y_j^{old}\right)$$

El caso práctico que se incluye en el artículo [2] toma como entradas las de la tabla 2.5.

Tras crear la capa de neuronas y normalizamos las entradas nos queda una representación como en la figura 2.5

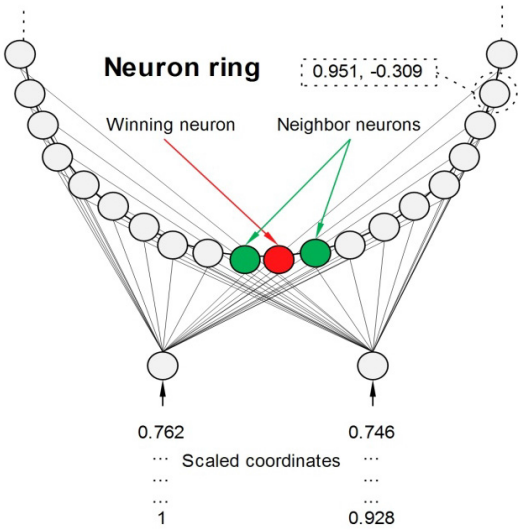

Figura 2.4: Representación de la arquitectura.

Con esto ya se puede realizar el aprendizaje, obteniendo un resultado parecido a la representación de la figura 2.6, donde se obtiene que el camino mínimo es $MTLKIADBOEQRSFHCGPJNM$.

WDS	latitude	longitude	WDS	latitude	longitude
A	53.214	19.155	K	52.349	18.924
B	53.560	19.221	L	49.763	19.076
C	53.280	19.167	M	52.758	18.790
D	53.111	19.220	N	52.988	18.920
E	54.200	19.210	O	53.719	19.194
F	54.390	19.185	P	53.684	19.031
G	53.848	19.079	Q	54.058	19.269
H	53.721	19.170	R	54.076	19.399
I	53.603	19.027	S	54.390	19.399
J	53.494	18.969	T	49.763	18.790

Tabla 2.5: Valores de entradas.

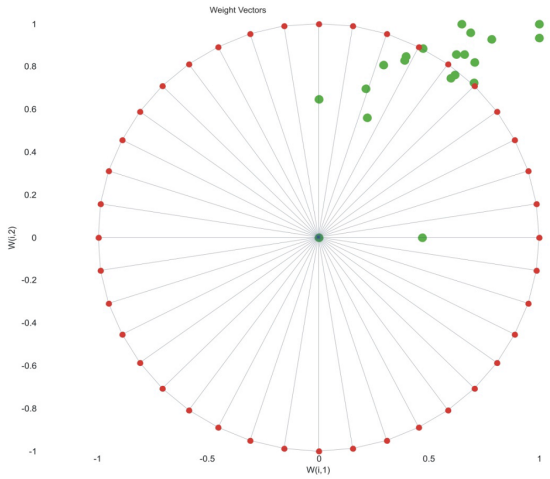

Figura 2.5: Neuronas y entradas normalizadas.

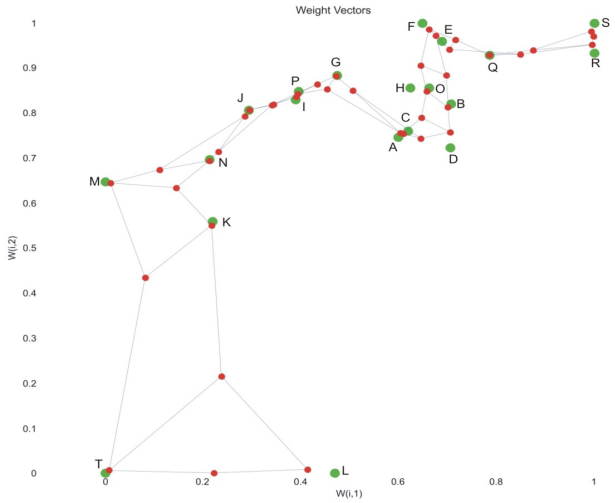

Figura 2.6: Neuronas y entradas normalizadas tras el aprendizaje.

Capítulo 3

Paralelización

La paralelización de procesos, como todo en esta vida, tiene sus ventaja y sus inconvenientes, en el caso que estamos estudiando, las ventajas de tiempo tienen mucho más peso que la dificultad que añade a la implementación. Esto se debe a que las RNA son extremadamente paralelizables, ya que consisten en unidades de proceso muy sencillas que interfieren entre sí en sus cálculos.

Es siempre necesario tener en cuenta el coste de la paralelización frente a la reducción posible por la paralelización.

En el caso de las adaptaciones al problema del viajante de comercio o TSM por sus siglas en Inglés, se ve que tiene mucho potencial. El único problema es que por cada entrada tenemos que actualizar tres elementos de la capa de neuronas, y también puede ocurrir, y ocurre, que hay entradas para las que la neurona ganadora es la misma.

3.1. Ventajas

La ventaja más notable es la mejora del tiempo, pero en el caso de las RNA la sencillez con la que se paraleliza, acaba siendo casi más intuitiva la implementación.

Matlab

En este caso las ventajas son numerosas, pero cabe destacar la facilidad con la que, con el uso del Parallel Computing Toolbox, se puede realizar numerosas aplicaciones sin mucho esfuerzo y sin necesitar un periodo de aprendizaje muy largo. Quizás más importante, es la oportunidad de usar un elevadísimo número de funciones que pueden ser paralelizadas sin más que indicar que se ejecuten en paralelo.

También es interesante mencionar la posibilidad de lanzar kernels de CUDA desde Matlab lo que nos da la posibilidad de usar toda la potencia de CUDA sin tener que cambiar de entorno de trabajo.

CUDA

Las ventajas de CUDA son también muy sorprendentes, el uso de una arquitectura diseñada de forma global por una misma organización, da lugar a un sistema muy óptimo. También es importe destacar las mejoras notables que ha transformado una arquitectura poco amigable, en una sencilla y bien documentada. Las mejoras incluyen la inclusión de un debugger con el NSight edition de las versiones de CUDA, compatibles con Eclipse y Visual Studio, también se ha simplificado la implementación mediante el uso transparente de la memoria para el programador a partir de CUDA 6.0, aunque este empeore el rendimiento y la inclusión de paralelismo dinámico, a partir de la versión 5.0, que consiste en la posibilidad de lanzar kernels desde un kernel, reduciendo el uso de CPU y transferencia de datos entre las memorias de CPU y de vídeo.

3.2. Inconvenientes

El inconveniente más notable en el caso de la utilización del procesador gráfico, es el peso que tiene lanzar código, tiene un coste elevado en tiempo. De esta manera tenemos que estar seguros de tener un problema que merece el esfuerzo y el coste vinculados a la paralelización.

Matlab

En este caso el principal problema es la facilidad con la que podemos llegar a caer en un caso de uso poco deseado para el rendimiento, y es el uso de funciones sin indexar en paralelo. Esto ocurre, por ejemplo, si tenemos un array alocado en la tarjeta gráfica, y en vez de usar alguna de las funciones para las operaciones elemento a elemento se utilice un bucle para operar sobre el array [3] en este artículo se incluyen ejemplos y banchmarking del uso de pagefun sobre arrays alocados en GPUs con funciones indexadas y no indexadas frente a los mismos cálculos en CPU.

CUDA

Hay dos inconvenientes principales para el uso de de CUDA, el primero es la curva de aprendizaje, este es muy empinado al principio, cosa que crea una barrera para los programadores acostumbrados a un paralelismo más intuitivo y en CPU. El segundo es, que al necesitar ejecutarse sobre un tarjeta NVidia, lo hace incompatible con muchos equipos, hace que a la hora de escoger tecnologías haya muchas empresas que aún usan tecnologías paralelas en CPU u optan por su contrapartida libre OpenCL.

3.3. Descripción de las tecnologías

Se han utilizado CUDA C/C++ utilizando Visual Studio community 2013 como IDE y Matlab junto con su Parallel Computing Toolbox.

CUDA

Como podemos leer en la misma página de NVidia, CUDA es una arquitectura de cálculo paralelo que aprovecha la gran potencia de la GPU

para proporcionar un extraordinario incremento en rendimiento. En este caso vamos a hacer uso de la capacidad de cálculo para calcular una red neuronal, para ello se ha usado la versión 6.5 de CUDA C/C++, y compilado con el compute capability 2.0, de tal manera, pudiéndose hacer uso de ciertas funciones desde los kernels de CUDA como pueden ser las impresiones de pantalla que da mucha facilidad a la hora de la implementación, y en particular la función de la raíz cuadrada, que en esta implementación es necesaria para el cálculo de la distancia entre los puntos.

Se ejecuta sobre una tarjeta gráfica NVidia GeForce GT 650M, con 384 núcleos y memoria dedicada de 2Gb.

Matlab

Para la implementación final se ha utilizado Matlab 14b junto con el Parallel Computing Toolbox, aunque para las pruebas de concepto se ha utilizado también el Neural Network Toolbox.

Parallel Computing Toolbox es una librería en Matlab que permite hacer uso de la capacidad de procesamiento de los equipos multicore, con tarjetas gráficas y en clusters.

Neural Network Toolbox es una librería de Matlab que incluye una serie de funciones y aplicaciones para modelar sistemas no lineales entre ellos aprendizaje por refuerzo, mapas auto-organizados y redes dinámicas.

Capítulo 4

Implementación

Antes de realizar la implementación se hacen una serie de pruebas de concepto, que se explicarán en detalle en el capítulo de los resultados, con los cuales se han tomado unas decisiones sobre la implementación, como puede ser el uso de memoria compartida en la implementación de CUDA o el uso del mismo kernel de CUDA en la implementación de Matlab.

4.1. Pruebas de rendimiento

Lo que se ha hecho para comprobar en un inicio la diferencia de tiempo entre usar Matlab con el Parallel Computing Toolbox, usar Matlab para ejecutar un kernel de CUDA, usar CUDA con la memoria compartida y CUDA con la memoria por defecto, se ha medido para comprobar la diferencia en la eficiencia de la implementación del uso de la tarjeta gráfica en Matlab, hice una prueba sencilla de lanzar un kernel mandándole un array de ceros de longitud 100, sumarle 1 a cada posición y devolver el resultado.

Matlab con Parallel Computing Toolbox
Esta solución hace uso de los gpuArray y de bsxfun para calcular la suma, se puede ver el script en la subcarpeta de Matlab en la carpeta de Prueba, llamándose sumaSimple.m.

Matlab lanzando kernel de CUDA
Esta solución lanza un kernel de CUDA ya compilado que calcula la suma, se puede ver el script en la subcarpeta de Matlab en la carpeta de Prueba, llamándose sumaKernel.m.

CUDA memoria por defecto
Esta solución lanza un kernel de CUDA usando la memoria por defecto para calcular la suma, se puede ver el script en la subcarpeta de CUDA en la carpeta de Prueba, llamándose sumaSimple.cu.

CUDA memoria compartida
Esta solución lanza un kernel de CUDA usando la memoria compartida para calcular la suma, se puede ver el script en la subcarpeta de CUDA en la carpeta de Prueba, llamándose sumaShared.cu.

4.2. Solución

Como ya se ha mencionado, la implementación que se ha realizado, es una adaptación del TSM en una SOM, para ello me he basado en las implementaciones descritas en el capítulo 2, con algunos cambios que mejoran el rendimiento.

Tras ver los resultados del capítulo 3, y teniendo en cuenta las pruebas de rendimiento estudiadas en el capítulo siguiente la implementación ha quedado como sigue.

Esta implementación usará como datos de prueba los usados en en la adaptación realizada para la ciudad de Nis [2].

Para poder paralelizar los cálculos creo una SOM donde la capa de sensores tiene tantos como número de entradas del mapa, M, y la capa de neuronas como un círculo centrado en 0 y radio 1, con N veces más neuronas que sensores, equidistantes, N es decidible en tiempo de ejecución, pero el ejemplo se realiza con $N = 3$. De esta forma podemos calcular el círculo de neuronas como un array de $N \times M$ coordenadas, se calcula como sigue:

angulo=360 $\cdot \frac{1}{M \times N}$

neuronas$(i,x) = cos\,(angulo \cdot i)\,\frac{\pi}{180}$

neuronas$(i,y) = sin\,(angulo \cdot i)\,\frac{\pi}{180}$

Tabla 4.1: Funciones para crear el círculo de neuronas.

Aplicando estas fórmulas al ejemplo de la ciudad de Nis obtenemos 60 neuronas dispuestas en círculo como en la figura 4.1.

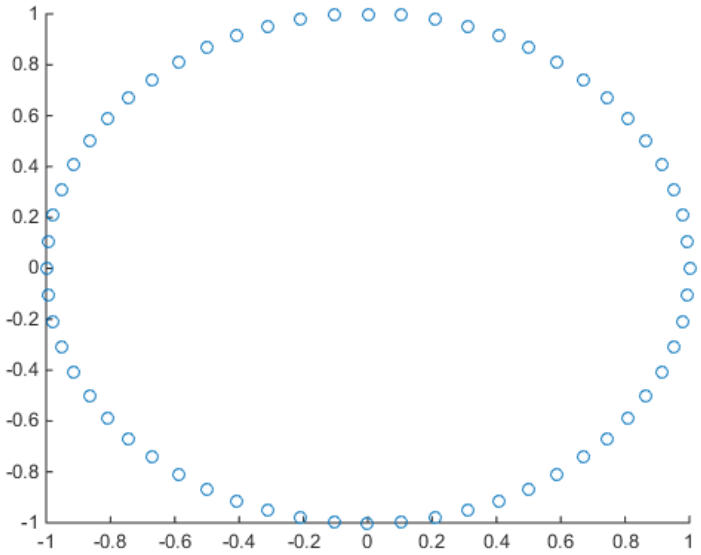

Figura 4.1: Representación del círculo de neuronas.

El siguiente paso consiste en normalizar las entradas, en esta implementación he cambiado la normalización para centrar los datos en 0 con las fórmulas incluidas en la tabla 4.2.

$$\text{input}(i,x) = input\,(i,x) - \frac{\left(minX+maxX)\cdot\frac{1,5}{2}\right)}{maxX-minX}$$

$$\text{input}(i,y) = input\,(i,y) - \frac{\left(minY+maxY)\cdot\frac{1,5}{2}\right)}{maxY-minY}$$

Tabla 4.2: Normalización de las entradas.

De esta manera, se obtiene una representación de sensores rodeados de un círculo de neuronas, como en la figura 4.2

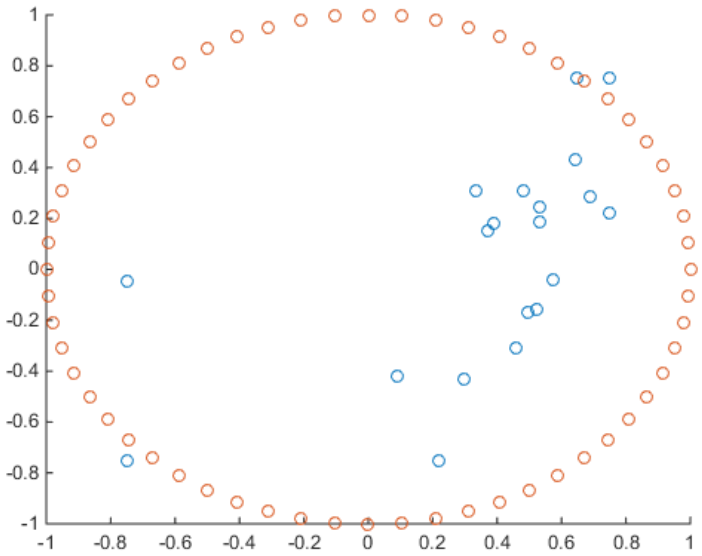

Figura 4.2: Representación los sensores normalizados y el círculo de neuronas.

En este caso no tenemos en cuenta los pesos sinápticos, de tal manera que el aprendizaje, al igual que en la adaptación para la ciudad de Nis, se hace sobre la capa de neuronas, actualizando su posición en cada iteración.

Para mejorar el número de iteraciones necesarias para el aprendizaje, he optado por dividir el aprendizaje en tres fases:

1. Para cada elemento de la capa de neuronas, se encuentra el sensor más cercano y se acerca a él la diferencia de coordenadas ponderada por la tasa de aprendizaje, en este caso 0.1, y por la frecuencia

relativa a la que ese input haya sido ganadora. En este ejemplo se ejecuta 50 veces.

2. Para cada elemento de la capa de sensores, se encuentra la neurona más cercana y se acerca a ella la diferencia de coordenadas ponderada por la tasa de aprendizaje, en este caso 0.1, en esta fase también se tienen en cuenta las neuronas 4 vecinas ponderando la actualización de las 2 vecinas inmediatas por 0.5, y las 2 vecinas a una de distancia ponderada por 0.25. En este ejemplo se ejecuta 100 veces.

3. Para cada elemento de la capa de sensores, se encuentra la neurona más cercana y se acerca a ella la diferencia de coordenadas ponderada por la tasa de aprendizaje, en este caso 0.5. En este ejemplo se ejecuta 400 veces.

Tras la ejecución nos encontramos con el siguiente camino mostrada en la figura 4.3, que coincide con el obtenido para la ciudad de Nis.

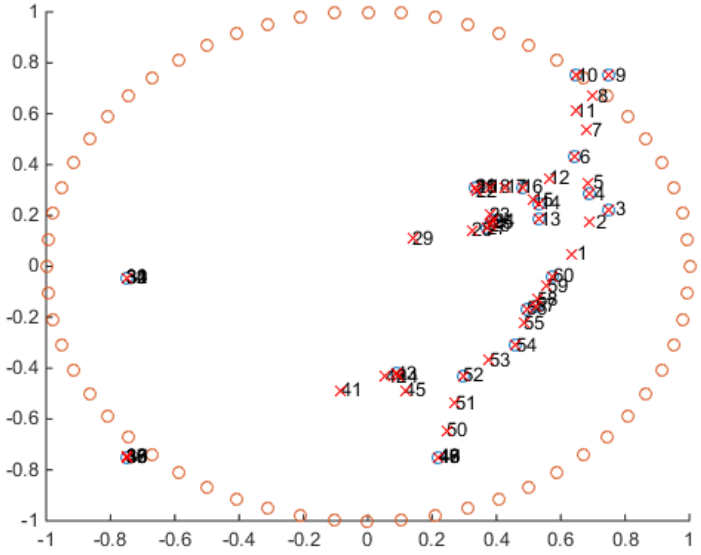

Figura 4.3: Representación del camino calculado superpuesto a los sensores normalizados y el círculo de neuronas.

Como se puede observar, el aprendizaje necesita 550 iteraciones para lograr el mismo resultado que la adaptación para la ciudad de Nis, que necesitaron 2000 iteraciones para conseguirlo, además, se obtiene la mejora de rendimiento de procesamiento paralelo frente a iterativo.

CUDA

En el caso de la implementación en CUDA se ha optado por usar la memoria compartida, ya que, como se puede observar en las pruebas de rendimiento descritas en el siguiente capítulo, la mejora es muy notable.

Matlab

En el caso de Matlab, el primer enfoque fue la de utilizar el SOM incluido en el Neural Network Toolbox ejecutado de forma paralela con el Parallel Computing Toolbox. Este acercamiento tiene dos problemas desde el primer momento, el primero, no se puede personalizar la disposición de las neuronas, y dado que esta implementación depende de ello hace imposible el uso del SOM. El segundo es que la implementación de Matlab del SOM no es soportado por el Parallel Computing Toolbox, ya que este utiliza funciones no soportadas como puede ser *negdist*.

El segundo enfoque consiste en implementar desde cero el algoritmo en Matlab utilizando el Parallel Computing Toolbox, pero con muy pocas iteraciones, esta implementación empieza a crecer en tiempo de ejecución llegándose a tardar casi un minuto para ejecutar sólo trescientas iteraciones. Para mejorar esta situación hay que sólo hacer uso de funciones indexadas, como se menciona en el capítulo 3, pero, estas funciones tienen unos casos de uso muy limitados en gran parte por que necesitan operar sobre dos arrays del mismo tamaño en el caso de bsxfun y sobre páginas del mismo tamaño en el caso de pagefun. Esto lleva a la utilización de arrays auxiliares, dónde se almacenan datos intermedios para igualar el tamaño de los arrays, pero esto tampoco mejora mucho el problema

del rendimiento, ya que, los datos utilizados crecen mucho deshaciendo la mejora por el uso de las funciones indexadas.

El tercer enfoque y el más óptimo, es la utilización de la funcionalidad que se encuentra en el Parallel Computing Toolbox para llamar a un kernel de CUDA desde Matlab, es además la solución más sencilla, ya que se reutiliza el mismo kernel que en la implementación de CUDA.

Capítulo 5

Resultados

Se han realizado una serie de pruebas de concepto descritos en el capítulo anterior para poder llegar a las implementaciones óptimas. En este capítulo además se repasan los resultados obtenidos tras la implementación en ambas tecnologías de una implementación del TSM con una red de Kohonen personalizada, las medidas de tiempo tomadas han sido de las ejecuciones propias del kernel incluyendo también la transferencia de datos entre memoria de CPU y de GPU en ambos sentidos, no se ha tenido en cuenta ni la lectura y escritura en ficheros ni la normalización de la preparación de los datos en la CPU, ya que no es de interés para el objetivo de este documento.

5.1. Eficiencia

En este apartado se expondrán los tiempos de ejecución de cada una de las pruebas y algoritmos implementadas con el fin de comparar los mejores resultados.

5.1.1. Pruebas de rendimiento

Todos los resultados expuesto en este apartado se realizan como media de 10 ejecuciones del código. Tal y como se explica en el capítulo anterior estas cuatro pruebas consisten en sumar 1 a cada elemento de un array de 100 posiciones.

Es importante mencionar, a raíz de estas pruebas, y respuestas de varias preguntas publicadas en varios foros además de la búsqueda en la documentación de Matlab, éste ejecuta el código en modo JIT, haciendo que, la primera ejecución de un código sea mucho más lenta que las posteriores.

Matlab con Parallel Computing Toolbox
Media de tiempo de la primera ejecución: 2096ms.
Media de tiempo de las posteriores ejecuciones: 0.9ms.

Matlab lanzando kernel de CUDA
Media de tiempo de la primera ejecución: 1623ms.
Media de tiempo de las posteriores ejecuciones: 5.8ms.

CUDA memoria por defecto
Media de tiempo de la primera ejecución: 685ms.
Media de tiempo de las posteriores ejecuciones: 282ms.

CUDA memoria compartida
Media de tiempo de la primera ejecución: 577ms.
Media de tiempo de las posteriores ejecuciones: 188ms.

5.1.2. Solución

CUDA
Media de tiempo de la primera ejecución: 1108ms.
Media de tiempo de las posteriores ejecuciones: 575ms.

Matlab
Media de tiempo de la primera ejecución: 2708ms.
Media de tiempo de las posteriores ejecuciones: 21ms.

Estos dos últimos resultados se cumplen cambiando las entradas, ya sea el mapa de entrada o los valores de la tasa de aprendizaje, también se mantienen de forma proporcional al número de iteraciones de aprendizaje, con lo cual, en cuanto a rendimiento, la mejor opción es la de integrar un kernel de CUDA en código Matlab, si no tenemos en cuenta el peso que tiene la ejecución del propio Matlab. Como ya se ha mencionado esto se debe a la ejecución JIT, también por ese factor podemos observar que la primera ejecución tarda más del doble con respecto a su contrapartida CUDA C/C++.

5.2. Calidad

La calidad es la misma en CUDA y Matlab, los valores obtenidos por ambas ejecuciones ha dado lugar a la misma solución óptima, que además coincide con la solución obtenida con la adaptación para la ciudad de Nis [2]

5.3. Dificultad

5.3.1. Pruebas de rendimiento

Matlab con Parallel Computing Toolbox
El uso de este Toolbox en Matlab, para las operaciones más comunes es extremadamente sencillo y muy asequible para cualquiera que tenga un conocimiento mínimo del uso de array en Matlab.

Matlab lanzando kernel de CUDA
La dificultad de lanzar un kernel de CUDA desde Matlab no tiene dificultad ninguna, es inmediato, sólo hay que saber crear arrays en GPU y seguir las instrucciones de la documentación del Parallel Computing Toolbox.

CUDA memoria por defecto
La inclusión de la transparencia de la memoria para el programador, simplificó en gran medida la curva de aprendizaje, haciendo esta arquitectura mucho más asequible para el programador medio.

CUDA memoria compartida
En cuanto a la dificultad en un primer instante, la implementación en CUDA parecía ser más compleja, pero tras un breve periodo de aprendizaje el uso de memoria compartida es muy sencilla respecto a la mejora en rendimiento que tiene asociada.

5.3.2. Solución

La dificultad de ambas implementaciones reside en la implementación del kernel, ya que aparte de esta funcionalidad hace poco más que leer y escribir de fichero.

La dificultad principal de esta implementación reside en decidir las fases de aprendizaje y los parámetros de entrada, ya que es muy fácil obtener resultados con un sobre ajuste, o una solución no satifactoria, ya sea por un sobre ajuste en una de las fases, o por que se excluya una de las entradas de la solución final.

Capítulo 6

Uso del software

En este capítulo daré una serie de pasos para poner en funcionamiento el software incluido en el cd.

En el cd viene incluido el código de la implementación de la solución al TSM con una adaptación de SOM incluida en la carpeta *Codigo* y el código de las pruebas de rendimiento en la carpeta *Pruebas*

6.1. Pruebas

En la carpeta *Pruebas*, encontramos dos carpetas *CUDA* y *Matlab* con el código de las pruebas de rendimiento hechas en CUDA C/C++ y Matlab respectivamente. Cada una de estas incluye una prueba que consiste en crear un array de ceros de cien posiciones en GPU, sumarle uno y volver el array a CPU.

6.1.1. CUDA

Para compilar este código, hay que ir a la documentación de CUDA, buscar el sistema operativo en el listado e instalar los paquetes necesario. Luego, siguiendo los pasos de la documentación hay que compilar los archivo *.cu en las subcarpetas.

En la carpeta *sumaSimple* podemos encontrar el código de la prueba de rendimiento para CUDA con la memoria por defecto y en la carpeta *sumaShared* el código de la prueba de rendimiento para CUDA con la memoria compartida.

Cuando estos se ejecuten, imprimirán por pantalla el tiempo que llevan en realizar la suma.

6.1.2. Matlab

Para ejecutar este código se necesita tener el Parallel Computing Toolbox.

Para ejecutar el código de *sumaSimple.cu* no se necesita ningún paso o configuración espacial, esta prueba de rendimiento es la que mide el tiempo para el uso de las funciones propias de GPU de el Parallel Computing Toolbox. Lo que imprime en el terminal de salida de Matlab es el tiempo que lleva la suma.

En el caso de *sumaKernel.cu*, tenemos que antes compilar el kernel *kernel.cu* también en esta carpeta, para ello, antes de ejecutar el script de Matlab por primera vez, hay que lanzar el siguiente código *system('nvcc -ptx kernel.cu')*, luego se ejecuta el script de forma normal. Lo que imprime en el terminal de salida de Matlab es el tiempo que lleva la suma.

6.2. Código

Como se puede observar, en la carpeta *Codigo* hay tres carpetas, dos de ellas referentes a las implementaciones en CUDA C/C++ y Matlab llamadas *CUDA* y *Matlab* respectivamente, y otra carpeta *Resultados* que contiene los resultados de las ejecuciones, los datos de entrada y un script en Matlab de visualización.

Antes de poder ejecutar el código hay que establecer el path donde se van a encontrar los datos de entrada y donde se van a escribir los resultados, este puede ser la misma carpeta Resultados. Esto se hace cambiando la ruta que hay en el archivo *path* al deseado dentro de las carpetas *CUDA* y *Matlab*, y es necesario que los datos de entrada se encuentren aquí.

Cuando se ejecutan ambas versiones de la implementación personalizada, la de CUDA y la de Matlab, saldrá una ventana para que el usuario introduzca los datos de compilación.

Esta ventana pedirá al usuario siete entradas:

1. nombre del archivo que contiene las coordenadas del mapa. Cada línea de este documento se compone de dos números separados por espacio estos corresponden a un punto. Para el ejemplo incluido hay que introducir *inputOrig.txt* este archivo tiene que estar en la carpeta indicada en el archivo *path*.
2. número de iteraciones de la primera fase de aprendizaje. Para el ejemplo incluido *50*.
3. suma del número de iteraciones de las fases dos y tres. Para el ejemplo incluido *500*.
4. número de la iteración de la suma anterior, a partir de la cual la segunda fase se convierte en la tercera. Para el ejemplo incluido *100*.
5. número de veces más grande que tiene que ser el círculo de neuronas con respecto al número de entradas. Para el ejemplo incluido *3*.

6. valor de la tasa de aprendizaje de las dos primeras fases. Para el ejemplo incluido *0.1*.

7. valor de la tasa de aprendizaje de la tercera fase. Para el ejemplo incluido *0.5*.

Tras coger los datos de entrada, el programa calcula un camino bueno para la entrada y lo guarda en una nueva carpeta dentro de la carpeta indicada en el archivo *path*. La carpeta nueva tiene como nombre $[\text{CUDA } \|MATLAB] - \{time\} - \{var1\} - \{var2\} - \{var3\} - \{var4\} - \{var5\} - \{var6\} - \{var7\}$.

6.2.1. CUDA

Tras compilar el código siguiendo la documentación de CUDA, y ejecutarlo, nos encontramos con una ventana como la de la figura 6.1.

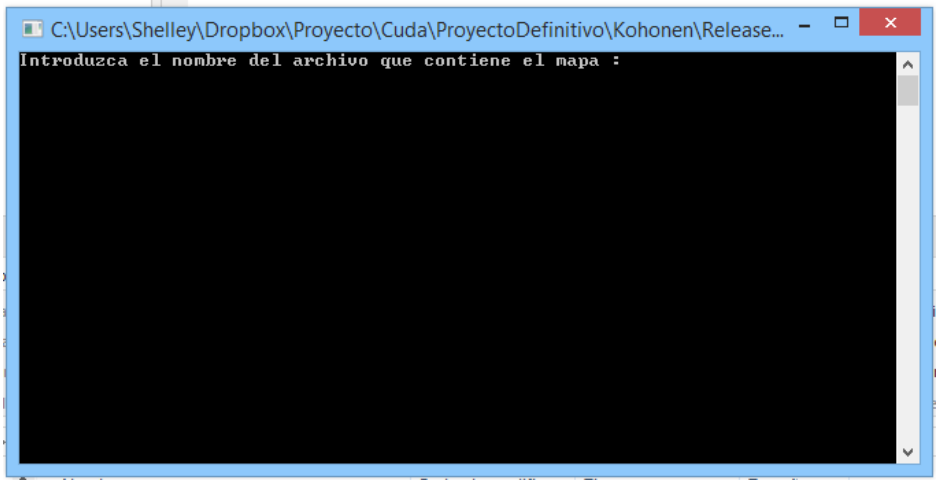

Figura 6.1: Ventana para la entrada CUDA.

6.2.2. Matlab

Para ejecutar este código hay que compilar primero el kernel *kohonen.cu* que se encuentra en la misma capreta *Matlab* con el siguiente comando *system('nvcc -ptx kernel.cu')*.

Ahora se puede ejecutar el script *kohonen.m* que lanzara una ventana como en la figura 6.2.

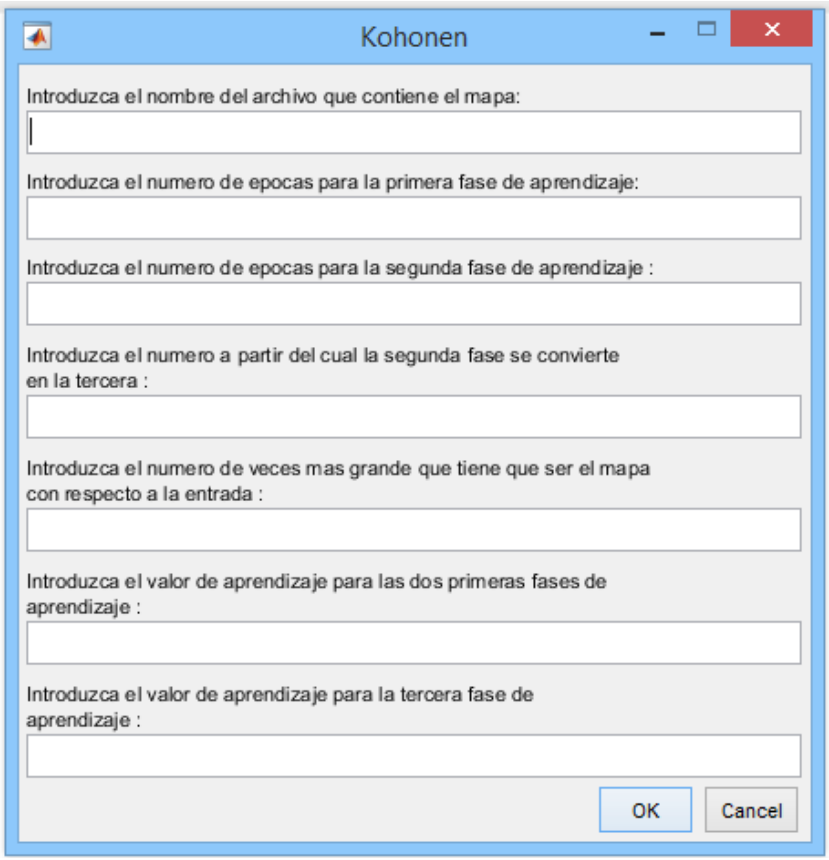

Figura 6.2: Ventana para la entrada Matlab.

6.2.3. Resultados

Para ver los resultados obtenidos por el software explicado anteriormente se lanza el script *drawMap.m* y este mostrará por pantalla una lista de

posibles soluciones, como en la figura 6.3. una vez seleccionamos una solución, este script mostrará por pantalla en una gráfica los sensores de entrada y el círculo de neuronas original como anillos de colores diferenciados y superpuesto el posicionamiento de las neuronal al final del aprendizaje como cruces rojas enumeradas como podemos ver en la figura 6.4

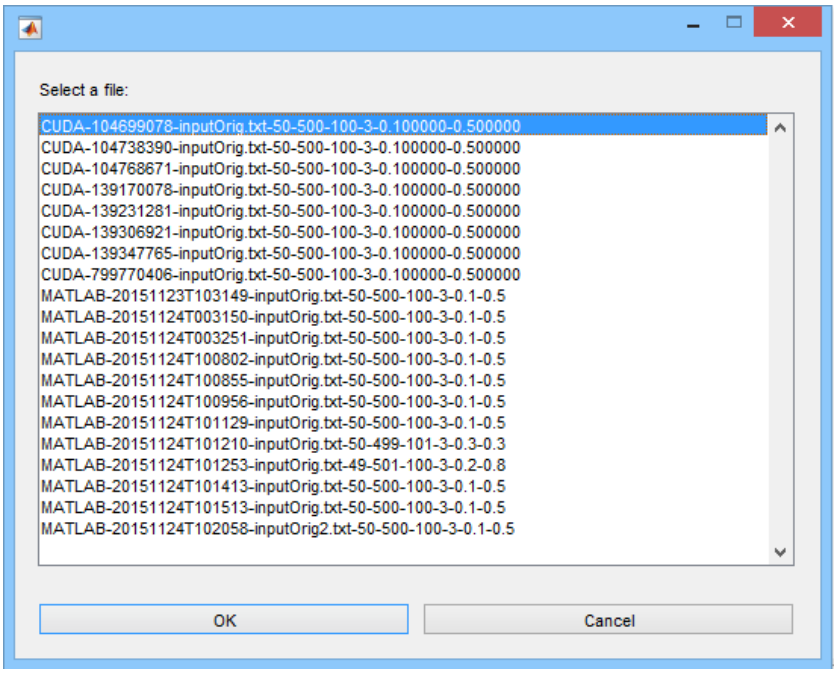

Figura 6.3: Selección de solución.

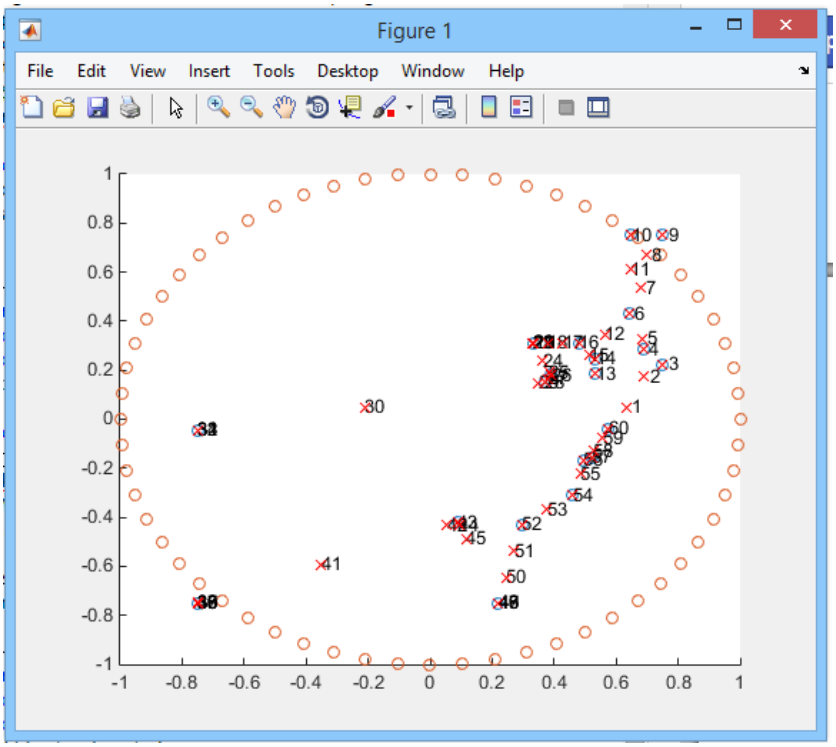

Figura 6.4: Gráfica de la solución.

Capítulo 7

Conclusiones

El uso de la arquitectura de CUDA da lugar a una eficiencia notable en la implementación de problemas paralelizables como puede ser el cálculo de un camino bueno para el TSM mediante el uso de SOM. Se consigue ejecutar de forma paralela el aprendizaje para cada entrada, reduciendo considerablemente el el tiempo de ejecución.

Como se ha expuesto en la sección de rendimiento del capítulo 5, la mejor opción es lanzar el kernel de CUDA desde Matlab, esto mejora el rendimiento, en las ejecuciones posteriores a la primera, frente a la implementación en CUDA C/C++ en un factor de casi 30. Esto además nos da la facilidad de usar las implementaciones tan intuitivas que el Parallel Computing Toolbox añaden, como puede ser la reserva de memoria en GPU. Además al lanzar un kernel, en este tenemos la oportunidad de optimizar los cálculos en la tarjeta gráfica, siendo un buen ejemplo, el uso de la memoria compartida, cosa que no se puede hacer en Matlab.

Las versiones de CUDA han ido mejorando la experiencia del programador, que llegados a este punto, en la versión 6.5, la implementación es muy intuitiva, además incluye soporte para muchas librerías de C o de C++, que añaden mucha funcionalidad a la arquitectura, también podemos notar las optimizaciones de tales librerías, ya que han sido ideados para tal uso.

El código resultante es muy legible e intuitivo, haciendo posible el trabajo en equipo y el mantenimiento de código durante un largo periodo de tiempo.

Bibliografía

[1] J. Muñoz, *Modelos Computacionales*. 2009-2010. Apuntes de clase.

[2] D. Markovic, M. Madic, V. Tomic, and S. Stojkovic, "Solving travelling salesman problem by use of kohonen selforganizing maps," *ACTA TECHNICA CORVINIENSIS - Bulletin of Engineering, Tome V*, October-December 2012.

[3] "Improve performance of small matrix problems on the gpu using pagefun," *Matlab Documentation 2015b*. http://es.mathworks.com/help/distcomp/examples/improve-performance-of-small-matrix-problems-on-the-gpu-using-pagefun.html.

[4] J. Sanders and E. Kandrot, *CUDA by Example: An introduction to General-Purpose GPU Programming*. 75 Arlington Street Suite 300, Boston MA 02116 USA: Addison-Wesley, 2011. Un libro con ejemplos de programación básica con CUDA.

[5] R. L. Spencer, *Introduction to Matlab*. Brigham Young University, 2000. Un libro con ejemplos de programación básica con MAtlab.